

# A Segment-Based DSM Supporting Large Shared Object Space

Benny Wang-Leung Cheung, Cho-Li Wang

The University of Hong Kong  
Department of Computer Science  
Pokfulam Road, Hong Kong  
{wlcheung, clwang}@cs.hku.hk

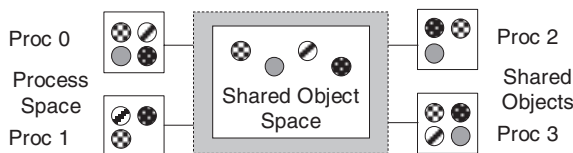
## Abstract

*This paper introduces a software DSM that can extend its shared object space exceeding 4GB in a 32-bit commodity cluster environment. This is achieved through the dynamic memory mapping mechanism, with local hard disks as backing store. We introduce the new concept of segments with intelligent splitting to reduce network traffic, false sharing as well as adapt better to the shared memory access patterns. A priority-based swapping algorithm is designed to reduce disk accesses for efficient dynamic memory mapping, and maximize the use of disk space as shared object space. A new queue-based scheme is also devised for efficient and simple management of memory blocks. The proposed solutions were implemented in LOTS V.2, and it can outperform its previous version when running small applications, while the maximum shared object space is increased to one-third of the total free disk space available among all the nodes.*

## 1. Introduction

Distributed Shared Memory (DSM) provides a paradigm for parallel programming, by offering a virtual view of globally shared memory among machines in the cluster, each of which can actually access its local memory (Figure 1). DSM is tempting to programmers because it handles the communication of shared data for application programmers, who do not need to insert explicit send and receive commands in the programs.

However, from day one software DSM was invented, virtually all DSM systems [3, 6, 9, 13, 19] had mainly put their focus on performance, while overlooked the



**Figure 1. The virtual shared address space provided by DSM (object-based view)**

lack of a large shared memory or object space. In particular, most DSM systems just map the whole shared memory or object space to the virtual memory space of every process during application execution using a fixed one-to-one mapping technique. As a result, the size of the shared object space cannot exceed the virtual memory space available for a single process, which is 4GB for 32-bit machines and operating systems. Moreover, as a large portion of the process space has to be reserved by the kernel, the program code, heap, stack and shared libraries, the actual size of the shared object space supported by the DSM systems will be far less than 4GB, regardless of the number of machines used. For example, when a cluster of PCs, each with 128MB RAM, is used to run TreadMarks [13], the most successful DSM system to date, only 96MB of shared memory can be created. Another DSM system, JIA-JIA, only allows 128MB of shared memory space to be claimed, so as to avoid performance degradation due to the invocation of the swap partition, in case the shared memory cannot be wholly mapped to the physical memory of each machine.

Such limitation and lack of scalability is clearly insufficient for the increasing demand for more shared memory by real-life scientific applications nowadays. For example, it is currently impossible to find the best move for a Go (Weiqi) game by pre-loading all possi-

This research is supported by Hong Kong RGC Grant HKU-7030/01E and HKU Large Equipment Grant 01021001.

ble chessboard configurations to the memory of a computer or cluster and then performing a simple lookup, since there are a total of  $3^{361}$  configurations. Even we can eliminate some repeated configurations due to symmetry, the number of unique configurations is still far larger than the number of addresses available in a 128-bit machine, if there exist one. Other applications, such as weather forecasting and DNA string matching also require tens of gigabytes of memory to execute. As a result, application programmers either need to resort to 64-bit clusters, which are yet to become the mainstream of commodity computers and thus cost substantially more, or they need to re-design the algorithms that consume less virtual memory. But this can take days or months to implement and the resulting program may execute considerably more slowly.

We believe that the provision of a large shared object space can allow a larger amount of complex scientific applications to enjoy the high cost-effectiveness of cluster computing and good programmability. So, we have developed the first version of LOTS [7], namely LOTS V.1, to support a shared object space larger than the virtual memory space of a process addressable by the underlying hardware, without any special hardware and compiler support. The local disk of each machine is used as the backing store for shared objects not being accessed when the local process space is full. Only a small amount of control information for each object is resident in the virtual memory, while the object data is dynamically but lazily mapped into the process space when being accessed. With such scheme, LOTS is able to provide a shared object space larger than the size of the process space. The swapping of shared objects between the process space and hard disk is done automatically during runtime, without the need of special code or dedicated compiler support. We have shown in [7] that the scheme is able to support hundreds of gigabytes of shared object space, with the only limit being the amount of free disk space available in each machine. Moreover, with the use of scope consistency model [10] and mixed coherence protocol, together with a few optimization techniques, DSM applications using small amount of shared memory can perform under LOTS as good as, or even better than traditional DSM when the hard disk needs not to be used as the backing store.

In this paper, we report our further improvements to LOTS V.1 in several aspects. The most important one is that the free space available in the hard disks of the participating machines is utilized more efficiently. With its new swapping protocol, we further expand the maximum shared object space supported by LOTS to become one-third of the total free disk space available in all machines.

In terms of performance, although LOTS V.1 is comparable to traditional DSM systems, it still contains much room for improvement. The transmission of large whole objects after synchronization, particularly barriers, can be very time-consuming. However, only a small part of the object may be updated or needed by a machine. In addition, the size of a large object such as an array can cause unnecessary false sharing – a major performance bottleneck in DSM systems – when different processes write to different parts of the object simultaneously. We have solved this problem by introducing the concept of variable-sized *segments*.

Finally, we successfully simplify the internal memory management scheme of LOTS. A new memory management scheme is devised by taking advantage of segments, hence reducing memory usage and making the swapping protocol more efficient.

All the above design enhancements have been implemented in the second version of LOTS, namely LOTS V.2. It is capable of providing an even-larger shared object space with better performance.

Section 2 of this paper will briefly introduce the LOTS project, addressing the features of the LOTS V.1 DSM system. Section 3 then discusses the potential problems of LOTS V.1 in detail, and describes the solutions as adopted in LOTS V.2. This will be followed by the testing and results analysis in Section 4. Section 5 presents the related work, and the paper will end with the conclusions and future work in Section 6.

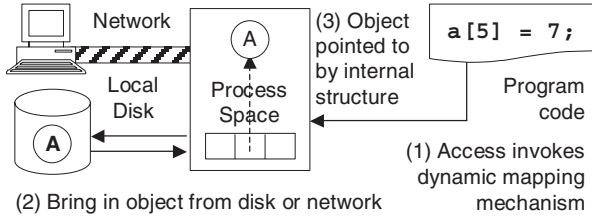
## 2 The LOTS DSM System

For the sake of completeness, we discuss the design of LOTS in this section. Interested readers may refer to [7] for a further description about the design and implementation details of LOTS V.1.

### 2.1. An Overview of LOTS

LOTS is implemented as a C++ runtime library on top of the Linux operating system. DSM applications making use of LOTS are compiled with the library using ordinary g++. As LOTS adopts an SPMD model, each participating machine will execute a copy of the application binary.

Figure 2 shows how LOTS can support a large shared object space without dedicated hardware and compiler support. When a process is going to access a DSM object, the runtime mechanism implemented in LOTS will be invoked through the use of C++ operator overloading facility, to check the status of the object, based on the control information saved at the process heap. If the object is not resided in the local process



**Figure 2. Illustrating the dynamic memory mapping mechanism in LOTS**

space, a copy will be brought in from the local hard disk. This action may also involve swapping out other unaccessed objects back to the disk if the region is full. We name this mechanism *dynamic memory mapping*.

## 2.2. Virtual Memory Management

LOTS allocates shared objects to an area outside the heap and stack of the process space to avoid contention while conforming to the memory management scheme imposed by the underlying operating system. This area is known as the *dynamic memory mapping* (DMM) area, since dynamic memory mapping can be performed on objects inside this area.

Due to this reason, shared objects are created through the use of a dedicated class provided by LOTS, instead of making use of the C++ `malloc()` or `new` functions. In particular, it uses a different allocation strategy compared with the Doug Lea’s allocator [18] for handling local objects. Just like the Doug Lea’s allocator, queues are implemented for storing the free memory blocks, which are arranged as doubly linked lists inside the DMM area. Each queue is designated for a different range of block sizes. However, another set of queues is needed in LOTS for managing the allocated memory blocks as well. This is because LOTS needs to select allocated blocks to be swapped out to the disk efficiently during the dynamic memory mapping process. LOTS V.1 uses 1024 queues to manage memory blocks of different sizes. Moreover, it allocates objects of small, medium and large sizes to the upper, middle and lower part of the DMM area respectively to improve access locality.

## 2.3. Dynamic Memory Mapping

When a shared object being accessed is not in the local DMM area, the *access checker* needs to bring in a clean copy, possibly swapping out a temporarily unused one during the process. The swapping algorithm can

greatly affect the overall performance and even the correctness of the DSM system. LOTS V.1 adopts a mix of best-fit and least-recently-used (LRU) swapping techniques. Timestamps are imposed on the shared objects, such that those last accessed within a certain threshold will not be swapped out. Other objects having the same size with the one to be accessed will have the highest chance to be swapped out. If no such object is available, larger ones will be considered next.

After the object is swapped into the DMM area, the LOTS access checker passes control to the *memory consistency manager*, which maintains the consistency of the shared object in access. If the local copy of the shared object is dirty, a clean copy or the updates in the form of *diffs* will be brought in from a remote machine. LOTS makes use of a mixed protocol to implement the scope consistency model. The protocol combines the homeless [8], write-update protocol for shared objects synchronized using locks, together with the migrating-home [5], write-invalidate protocol for shared objects synchronized by barriers. This can reduce the number of messages, and in particular, eliminate the possible traffic congestion that may arise after a barrier. Moreover, to eliminate the diff accumulation problem [17] suffered by most DSM systems when diffs are sent during synchronization, the *per-word timestamp* technique, that is, a timestamp associated with every four bytes of the shared object, is adopted.

We have shown in the testing results of [7] that this combination of protocols is very competitive, as LOTS V.1 performs as good as or even better than another DSM traditional system JIAJIA [9] in executing programs with small demand of shared memory.

## 3 LOTS V.2

In this section, we address several space and time performance issues in LOTS V.1, and discuss the corresponding solutions in the new version, LOTS V.2.

### 3.1 Unit of Shared Memory

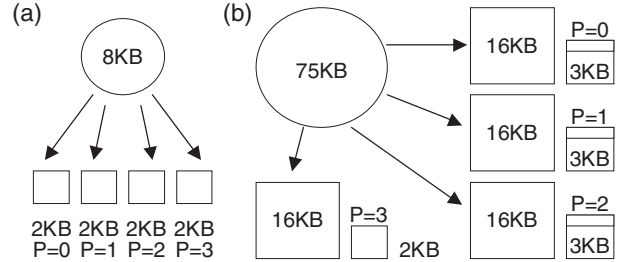
In LOTS V.1, the basic unit of shared memory is objects. The object-based nature has an advantage over the page-based counterparts in that small objects, such as those with primitive types and small arrays, will not suffer from the false sharing phenomenon as easily as page-based DSM systems do. However, when a coherence protocol is used with the concept of home, such as the migrating-home protocol [5], and a process has a dirty local copy of a shared object, the protocol requires that the whole clean copy of the object has to be sent to the local process from the home processor

before the object is accessed. If the object is very large but only one small part of the object is going to be accessed, a large amount of data traffic will be invoked and most of the traffic becomes redundant. Worse still, if there is another process requesting some data from the local process while it is receiving a copy of a large shared object, it will have to wait so long that the flow control may mistakenly regard the request as lost during communication. The unnecessary message retransmission further degrades the overall performance.

LOTS V.2 solves the above problem by introducing the notion of *segments*, which is a portion of a shared object. A segment will not exceed 16KB, so that the segment, its twin and the DSM control information can be sent altogether in a single message within the 64KB maximum limit posed by Berkeley sockets. With the use of segments, a process accessing a dirty shared object will receive the appropriate segment in a single message, hence reduces unnecessary data traffic and eliminates possible blocking by a long message. Moreover, since each message now contains the data of an entire segment, message assembly and disassembly to obtain the whole object becomes unnecessary, and the receiving process needs not wait for all messages comprising the same object to arrive before assembling them. Memory usage is also reduced.

Segments also help to reduce false sharing, since in object-based DSM, multiple processes writing to different parts of the object (such as an array) will be considered a kind of false sharing. With segments, processes can freely write to different segments of the same object simultaneously without suffering from false sharing. Unless multiple processes write to different parts of the same segment at the same time, which rarely happens due to the relatively small size of segments, false sharing will not occur in LOTS V.2.

To improve performance further, LOTS V.2 tries to split an object intelligently into  $N$  equal-sized segments, with  $N$  equal to a multiple of the number of processors  $P$  used. Each processor will then be assigned home of  $N/P$  consecutive segments, in the hope that the home assignment matches the access pattern of the object well, as shown in Figure 3. In case the prediction is incorrect, the penalty will still be minimal due to the presence of the migrating-home protocol. After the first barrier synchronization, the home of a shared segment will be migrated to one of the last processes accessing the segment, so that a higher chance of adapting to the memory access pattern can be obtained. A token approach is adopted just like LOTS V.1, except that a token is associated with each segment instead of each object. Tokens can be transferred during access of a segment, and the process holding the token of a



**Figure 3. The intelligent splitting algorithm from objects to segments in LOTS V.2**

segment at barrier synchronization time becomes the new home of the segment.

One more advantage of using segments in LOTS V.2 is that only the segment going to be accessed will be dynamically mapped to the DMM area. Other segments of the same object can still reside at the secondary storage or peer memory. In contrast, LOTS V.1 requires the whole object to be dynamically mapped. Thus segments allow faster swapping and help preserve better temporal locality, as a smaller size of shared memory will be swapped in and out each time. Moreover, unlike LOTS V.1, the maximum size of a single shared object in LOTS V.2 will not be limited by the largest free contiguous block available in the DMM area.

### 3.2 DMM Area Management

As mentioned in Section 2.2, LOTS V.1 allocates shared objects of different sizes in different parts of the DMM area. This is complicated as LOTS V.1 has to deal with every possible memory block size. In particular, if a large shared object is going to be allocated but the lower half of the DMM area is full, it has to be mapped to the upper half, which is intended for small objects. The result is a mix of small and large objects, which violates the intension of the original design. As we shall discuss later, the swapping process will cause further troubles to the DMM area.

As LOTS V.2 introduces segments of at most 16KB in size, the DMM area can be re-designed for simpler and more efficient management. LOTS V.2 divides the DMM area into 13 different sizes of slots from 4 bytes to 16KB. Each slot is used to store a shared segment of the closest size smaller than the slot. Slots of the same size will be adjacent to each other in the DMM area. Objects of 2KB or below can still benefit from the possible access locality, since adjacent small objects will be allocated and share the same physical page.

The number of queues used in LOTS V.2 can also be substantially reduced. Instead of 1024 queues as

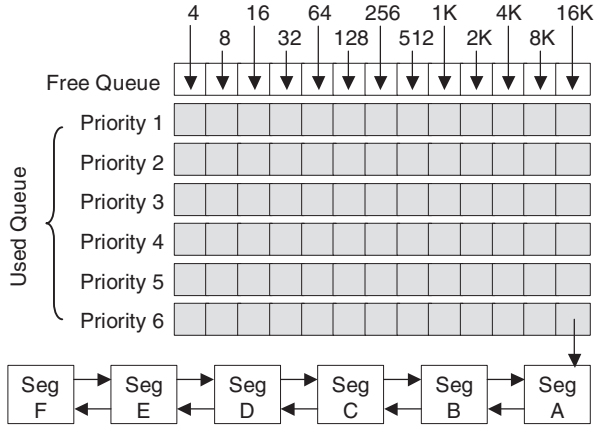


Figure 4. Queue management in LOTS V.2

needed in LOTS V.1, only 13 free queues are implemented, one for each size of free blocks. In addition, six used queues are provided for each size of used blocks, making a total of 78 used queues, as they are used for supporting a more efficient swapping algorithm in LOTS V.2 as described in the next subsection.

As shown in Figure 4, blocks of the same size and type are still managed as doubly linked lists, with the head of the list pointed to by the corresponding queue entry. But there is a change in the design of a memory block in the DMM area. Instead of putting the control information of a block, including the block size and the previous and next pointers, into the header and trailer of the block, LOTS V.2 decides to put this information into a new data structure called the *slot map* inside the heap. As all control information of all mapped slots in the DMM area are put together in the slot map that spans consecutive pages, the swapping algorithm can become faster as the traversal of the doubly linked list will involve fewer memory pages.

### 3.3 New Swapping Algorithm

LOTS V.1 adopts a combination of best-fit plus LRU algorithm to determine which object is going to be swapped out to make space for the object in access. This algorithm becomes space inefficient when no objects of the same size as the one in access can be found in the DMM area. A larger object has to be swapped out, and this often results in external fragmentation of the DMM area. Worse still, if the object in access is the largest shared object, the algorithm needs to swap out multiple smaller objects in order to empty enough space. With the introduction of segments in LOTS V.2, the algorithm can be greatly simplified, as we can always find a slot with the same size as the one in

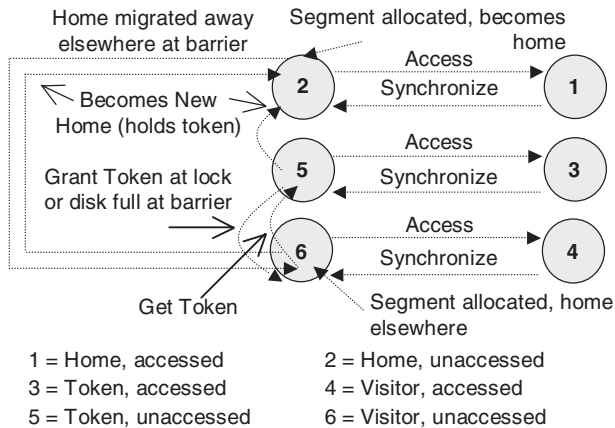
	<i>Accessing</i>	<i>Non-accessing</i>
Home	1	2
Token	3	5
Visitor	4	6

Table 1. Priority of different types of segments to be retained in the DMM area

access for swapping, and there will not be any external fragmentation. To further enhance performance, LOTS V.2 utilizes the runtime access information of each shared segment to determine which segment is to be swapped out, to reduce the number of disk accesses and improves the overall performance.

In LOTS V.2, the shared segments are classified into six types during runtime by each executing process, according to their current access status. At any time, a segment can have one of the three states in any process  $p$ : *home*, *token-owner*, and *visitor*. Home means the segment is the current home at process  $p$ , and can be migrated during barrier synchronization. The segment is considered token-owner when it is not home, but it obtains the token so that it can become the new home after the barrier. All the other non-home processes of the segment without holding the token are known as visitors. Besides the three states, the segments are further divided into accessing and non-accessing ones, which reflect whether they have been accessed after the last synchronization point (lock release or barrier). Together the segments are classified into six types, and a priority is assigned to each of them as shown in Table 1. A smaller number means a higher priority for the segment to be retained in the DMM area.

Such priority is decided according to the rationale below. Home segments always have the highest priority to be retained in the DMM area, since the home process has to keep the most updated copy of the segment, and it has to serve all requests for a copy of the segment. Retaining them in the DMM area can reduce the disk access overhead. On the other hand, visitor segments without being accessed after synchronization are the safest candidates to be swapped out, since there is no indication that they will be accessed in future. For other non-home segment types, an accessing segment is sure to be accessed again at the next synchronization point for calculating the diffs [12]. So it should have a higher priority to be retained in the DMM area than a non-accessing segment. Note that the status of each segment can be changed dynamically during DSM execution. Figure 5 shows the state transition diagram among the six states.



**Figure 5. State transition diagram of the six different segment states in LOTS V.2**

The final new decision made by LOTS V.2 concerns where the segment being swapped out is to be accommodated. In LOTS V.1, the victim segment, together with the twin and the per-word timestamps are always written back to the local hard disk. This causes the size of the shared object space that can be supported by LOTS V.1 no more than the minimum disk space available among all the participating machines. If one of the machines executing a DSM application under LOTS V.1 has less free space in its hard disk, other machines will be affected as well, hence lacking scalability. In LOTS V.2, a non-home victim segment is allowed to send back to the home node for storage. The updates made will be sent, just like at synchronization point. This helps to preserve memory consistency. Performance will not degrade though, since the network transmission overhead is comparable or even smaller than the hard disk access overhead, especially if a high-speed cluster network is used. In case too many segments are migrated to a process, such that the total size exceeds the available disk space, home migration will be forbidden. Since the local hard disk is used to store the home segments only, the new swapping mechanism allows the maximum shared object space supported by LOTS V.2 to increase to one-third of the total free disk space in all the machines executing the DSM application, as each segment of size  $S$  needs  $3S$  disk space storage, one for the twin, one for the per-word timestamps and one for the segment itself.

## 4 Testing and Results

In this section, we present the execution timing results for LOTS V.2 for two types of applications. In

the first part, LOTS V.2 is executed with applications with small problem size, hence needing a small amount of shared memory. LOTS V.2 behaves just like traditional DSM systems when the large object space support is not invoked. The execution time is compared with both the first version of LOTS V.1 and JIAJIA V1.1, a page-based software DSM employing the home-based protocol to implement scope consistency. In the second part, large applications with a high demand of shared memory is executed in LOTS V.2. Instead of the execution time, we focus on the size of shared object space that can be supported, as well as identifying how the amount of RAM in each machine of the cluster can affect the performance of the system.

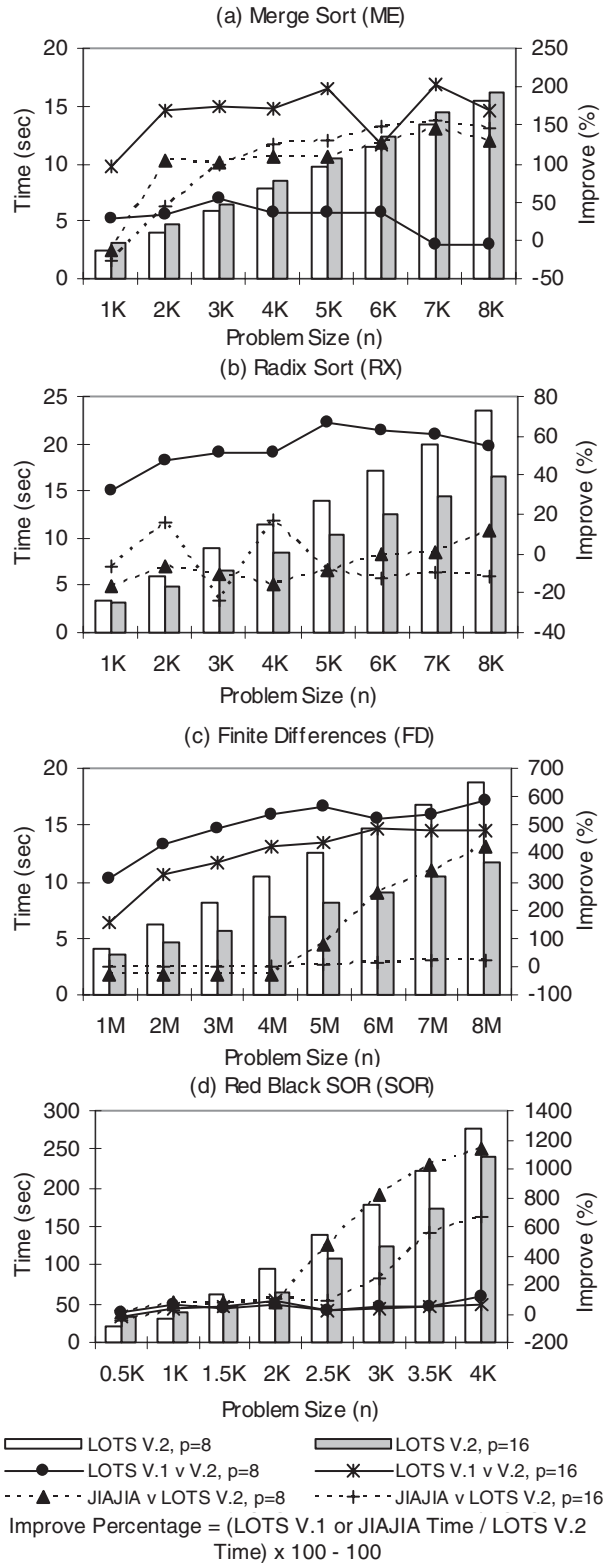
### 4.1 Small Applications

For the first test, we have executed four different applications, each with eight different problem sizes and two different node sizes, under JIAJIA, LOTS V.1 and LOTS V.2. The applications include ME (merge sort), RX (radix sort), FD (finite differences with 20 iterations) and SOR (red-black successive SOR with 256 iterations). Except FD, the other applications have been used in the testing of [7]. The testing was performed on a 8- and 16-node Pentium IV 2GHz PC cluster, connected by a 100Mb/s Ethernet switch. Each PC has 512MB RAM and 2GB swap space, and Linux Fedora Core 1 was used as the operating system.

The timing results are shown in Figure 6(a) to 6(d). The bars show the execution time of each application with different problem sizes and nodes under LOTS V.2. Note that it takes longer to run ME when 16 processes are used instead of 8. This is because the numbers in each process are already sorted during initialization. With the number of processes doubled, one extra stage is needed to merge the numbers together.

On the other hand, the lines show the improvement percentage of LOTS V.2 over LOTS V.1 or JIAJIA V1.1. A percentage above 0 means that LOTS V.2 executes faster than the other DSM. The larger the value, the more efficient LOTS V.2 is over its counterpart.

The performance results suggest that LOTS V.2 is more efficient than JIAJIA V1.1, especially for large problems. In particular, for FD and SOR with 8 processes, LOTS V.2 can execute 4 and 12 times faster than JIAJIA V1.1 respectively for the largest problem size tested. This is because most shared variables are accessed by only one process throughout the execution of FD and SOR. The migrating-home protocol in LOTS V.2 can bring them to the accessing process after the first iteration, hence reducing network traffic in subsequent iterations. ME also benefits from the migrating-



**Figure 6. Execution time of LOTS V.2 on the four applications with small memory demand**

home protocol, as it adapts well to the reverse tree-based access pattern for shared objects. However, for RX, most shared variables are accessed alternatively by two processes. Such access pattern does not favor the migrating-home protocol, as the home will be migrated to-and-fro between the two processes.

LOTS V.2 can also outperform the first version as well. Note that for each of the four applications, we have declared a two-dimensional shared array to store the numbers in LOTS V.1, although a one-dimensional shared array is more natural in implementing ME, RX and FD. The performance of the two-dimensional array version is much faster than the one-dimensional one, because LOTS V.1 treats a one-dimensional array as a single object. False sharing becomes very serious when different processes write to different parts of the array at the same time. On the other hand, in LOTS V.2, a one-dimensional array is declared for each application, so as to avoid the doubled overhead for operator overloading during shared object access. In other words, we are comparing the best performance achievable by both versions of LOTS, and LOTS V.2 beats its predecessor for most data points. This is attributed to the introduction of segments with intelligent splitting strategy. The one-dimensional shared array is automatically splitted into different small segments no more than 16KB in size. They reduce the redundant network transmission overhead for maintaining memory consistency, and adapt well to the shared memory access patterns of DSM applications by reducing the degree of false sharing.

## 4.2 Testing Large Object Space Support

As LOTS is the first DSM system to support a large shared object space, we are more keen on studying the performance of applications that demand a large amount of shared memory, preferably more than 4GB. We tested LOTS V.2 with the use of three real-life applications. SOR has been used in the previous section, while two new applications, namely MA (matrix addition) and MT (matrix transpose) were introduced.

We performed the test in two different environments. The first one was the Pentium IV cluster as discussed in Section 4.1. Each of the machines was equipped with a hard disk with 9GB of free space. The other testing platform was a cluster of eight Pentium III 700MHz machines, in which each machine has 2GB of RAM and about 6GB of free hard disk space. Through testing in two different environments, we can have an idea of whether a fast CPU or more main memory is more vital to the performance of LOTS V.2.

The execution results are shown in Table 2. We can

<i>Problem Problem</i>	<i>Size (n)</i>	<i>Proc (p)</i>	<i>Total Shared Object Size</i>	<i>Eff. Shared Object Size</i>	<i>P3 Cluster Exec Time (s)</i>	<i>P4 Cluster Exec Time (s)</i>
MA	32K	4	4GB+128KB	12GB+384KB	19208	53579
MA	32K	8	4GB+128KB	12GB+384KB	25046	61000
MT	32K	4	4GB+128KB	12GB+384KB	9551	15861
MT	32K	8	4GB+128KB	12GB+384KB	3538	9234
MT	64K	8	16GB+256KB	48GB+768KB	22924	35678
SOR	32K	4	8GB+256KB	24GB+768KB	54491	100861
SOR	32K	8	8GB+256KB	24GB+768KB	35599	55645

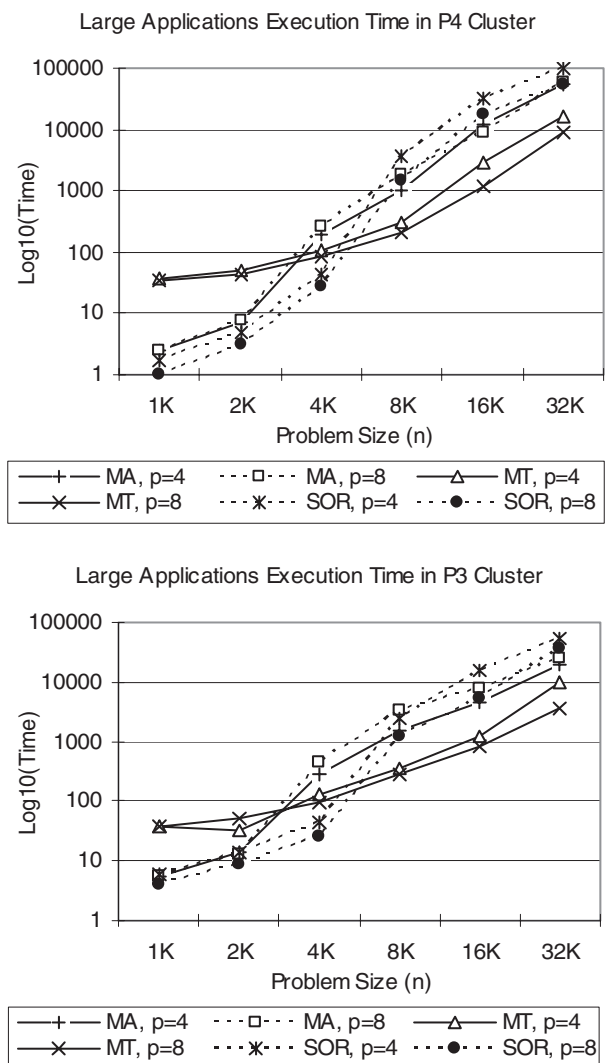
**Table 2. Performance of applications with large shared object space support**

observe that LOTS V.2 is able to support a shared object space of larger than 4GB. Note that due to the presence of the twin area and the DSM control area, the effective shared data size claimed by LOTS V.2 is actually 3 times the size of the shared object space. Another observation is that when 8 machines are used, LOTS V.2 is able to execute the MT application with problem size  $n = 64K$ . This amounts to a total shared object space of 16GB, which is exactly one-third of the total free disk space available in the Pentium III cluster. This verifies our claim that LOTS V.2 is able to support a maximum shared object space of 1/3 of the total free disk space available in the participating cluster. If larger hard disks are available, LOTS V.2 can support a shared object space of hundreds of gigabytes. Next, by comparing the execution time using the two clusters, we find that the Pentium III cluster has a better execution time than the Pentium IV cluster, despite slower CPUs are in use. The reason is attributed to the presence of more RAM in the Pentium III cluster, which significantly reduces the activities of swap partition provided by the underlying operating system.

Finally, when we plot the execution time of the three applications with increasing problem sizes, we obtain the graphs in Figure 7. The results show that the execution time is not directly proportional to the problem size. This is because when the problem size increases, the large shared object space support of LOTS V.2 is invoked and hard disk accesses become more frequent. This results in extra overhead and the execution time increases more than proportional.

## 5 Related Work

The need for a large shared object space has not been addressed by previous DSM researches. One way to achieve an object space larger than the process space addressable by the underlying hardware is *pointer swizzling* [21]. It refers to the runtime translation from



**Figure 7. Large applications execution time with increasing problem size**

system-assigned object identifier known as *persistent pointer* to virtual memory address called *transient pointer*. In the translation process, *persistent objects* resided in the disk will also be brought into the process space, while unused objects in the virtual memory can be temporarily removed (unswizzled) to the disk to make space for the others. However, this approach has never been adopted in DSM researches. Persistent storage systems such as QuickStore [20], ObjectStore [15] and Thor [4] did apply this technique. But instead of providing a large object space, they adopted various optimization strategies [14] to provide faster access of persistent pointers by avoiding multiple references from performing excessive amount of table lookup.

The mechanism used to achieve a large shared object space in LOTS V.2 is in some sense analogous to pointer swizzling. During dynamic memory mapping, LOTS brings in a segment from the disk to the local process space, involving a translation from persistent address (segment ID) to transient address (virtual memory address). However, LOTS does not aim at reducing the table lookup overhead. Instead, multiple references from one segment to another still require multiple table lookup, as the pointers store the segment IDs rather than the destination virtual memory addresses. LOTS opts not to adopt any optimization because unlike a persistent storage system, a DSM needs to keep track of the access states of every segment. These operations require table lookup, which waters down the benefits of pointer swizzling optimizations. Furthermore, LOTS uses operator overloading to trigger the address translation while the persistent storage systems aforementioned require the cooperation of dedicated compiler support and the page-faulting mechanism of the underlying operating system.

The segment concept as introduced in Section 3.1 shares the same name with those proposed in [1, 2, 16], but their ideas are very different. In these literatures, the term “segment” all refers to a collection of multiple pages as a logical memory consistency unit, while a segment in LOTS V.2 may be smaller than or larger than a page, ranging from 4 to 16K bytes. The actual size of a segment depends on the size of the original object, as well as the number of processes used.

The segments in LOTS V.2 also exhibits certain differences from previous researches to provide a fine-grained DSM. Examples are the notion of regions in Jackal [19] and CRL [11]. These systems provide a fine granularity of sharing solely to enhance the performance by avoiding the problem of false sharing. However, segments in LOTS V.2 also aim at reducing the transmission of the updated copy through the network. This is because in a large shared object space, objects

tend to be large too. Sending large objects as a whole becomes a major performance bottleneck.

In addition, the size of segments in LOTS V.2 is determined intelligently by the system. With the number of executing processes in consideration, LOTS V.2 splits an object into segments to try matching the shared memory access pattern. Such a strategy has never been adopted in other DSM systems. In Jackal, regions can only be applied to arrays, with each region fixed at 256 bytes, regardless of the size of the object. The home of each region is not allowed to migrate to other processors throughout program execution (i.e., a home-based protocol). CRL allows variable-sized regions to be created, but the size of each region needs to be defined by the application programmer during shared memory allocation. This requires the programmer’s experience to set the correct region size for optimal performance. The splitting of objects to segments by LOTS V.2 is purely automatic, hence preserving good programmability while adapting well to many common shared memory access patterns.

## 6 Conclusion and Future Work

This paper introduces LOTS V.2, a second version of the DSM system LOTS, for supporting a large shared object space greater than 4GB on 32-bit clusters for real-life scientific applications. While LOTS V.1 is an object-based DSM, LOTS V.2 introduces the concept of segments, which have many benefits over objects as the basic unit of the DSM system. The smaller size of segments allows less data to be transmitted through the network or hard disks, avoids I/O blocking, reduces false sharing, simplifies the internal memory management, eases message handling, and makes dynamic memory mapping more efficient. In addition, LOTS V.2 introduces intelligent splitting of segments from objects, in order to adapt better to most of the common shared memory access patterns. LOTS V.2 also devises a new priority-based swapping strategy based on the access status of segments. This allows more efficient use of disk space for supporting a larger shared object space than the previous version.

Apart from these new features, LOTS V.2 preserves most of the ideas from LOTS V.1. It implements the dynamic memory mapping mechanism using a pure user library approach with an off-the-shelf compiler, without any dedicated compiler, preprocessor or operating system modification. Good programmability is also maintained, while the use of the migrating-home based coherence protocol allows LOTS V.2 to perform reasonably better than many traditional DSM systems, which are only capable of executing small applications.

From testing results, we have proved that LOTS V.2 is capable of executing applications with high demand of shared memory. It better utilizes the free hard disk space from the machines to support a larger shared object space. Due to the presence of segments, false sharing is reduced and thus the performance is better than LOTS V.1. However, the translation overhead from objects and offsets to segments at runtime is still high, especially when a multiple-dimensional array is accessed. We believe the performance can be further improved if such translation can be made more light-weighted. In addition, swap partition activities triggered by the operating system can be a major source of performance bottleneck in supporting a large object space. While the testing results indicates that LOTS V.2 runs better on clusters with more RAM, in the long run, we have to find out ways in order to reduce the impact of the operating system swap activities.

## References

- [1] R. Ananthanarayanan, S. Menon, A. Mohindra, and U. Ramachandran. Experiences in integrating distributed shared memory with virtual memory management. *Operating Systems Review*, 26(3):4–26, July 1992.
- [2] A. Banerji, D. Kulkarni, J. Tracey, P. Greenawalt, and D. L. Cohn. High-performance distributed shared memory substrate for workstation clusters. In *Proc. of the Second IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-2)*, 1993.
- [3] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. *CMU-CS-91-170*, pages 528–537, Sept. 1991.
- [4] M. Castro, A. Adya, B. Liskov, and A. C. Myers. Hac: Hybrid adaptive caching for distributed storage systems. In *Proc. of the ACM Symposium on Operating System Principles (SOSP'97)*, Saint-Malo, France, Oct. 1997.
- [5] B. W. L. Cheung, C. L. Wang, and K. Hwang. A migrating-home protocol for implementing scope consistency model on a cluster of workstations. In *the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, Las Vegas, Nevada, USA, June 1999.
- [6] B. W. L. Cheung, C. L. Wang, and F. C. M. Lau. Building a global object space for supporting single system image on a cluster. *Annual Review of Scalable Computing*, 4, 2002.
- [7] B. W. L. Cheung, C. L. Wang, and F. C. M. Lau. Lots: A software dsm supporting large object space. In *Proc. of 2004 IEEE Int'l Conference on Cluster Computing (Cluster2004)*, pages 225–234, Sept. 2004.
- [8] A. L. Cox, E. de Lara, C. Hu, and W. Zwaenepoel. A performance comparison of home-less and home-based lazy release consistency protocols in software shared memory. In *Proc. of the 5th High Performance Computer Architecture Conference*, Jan. 1999.
- [9] W. Hu, W. Shi, and Z. Tang. Jiajia: An svm system based on a new cache coherence protocol. In *Proc. of the High-Performance Computing and Networking Europe 1999 (HPCN'99)*, pages 463–472, Apr. 1999.
- [10] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA '96)*, pages 277–287, June 1996.
- [11] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: High-performance all-software distributed shared memory. In *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, volume 29, pages 213–226, 1995.
- [12] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.
- [13] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwanepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, Jan. 1994.
- [14] A. Kemper and D. Kossmann. Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis. *VLDB Journal*, 4(3):519–566, July 1996.
- [15] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Communications of the ACM*, 34(10), Oct. 1991.
- [16] I. Lipkind, I. Pechtchanski, and V. Karamcheti. Object views: language support for intelligent object caching in parallel and distributed computations. *ACM SIGPLAN Notices*, 34(10):447–460, 1999.
- [17] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proc. of SuperComputing'95*, Dec. 1995.
- [18] A memory allocator by doug lea. <http://gee.cs.oswego.edu/dl/html/malloc.html>.
- [19] R. Veldema, R. A. F. Bhoedjang, and H. E. Bal. Jackal, a compiler based implementation of java for clusters of workstations. In *PPoPP 2001*, 2001.
- [20] S. J. White and D. J. DeWitt. Quickstore: A high performance mapped object store. In *ACM SIGMOD International Conference on Management of Data*, pages 395–406, Paris, France, May 1994.
- [21] P. R. Wilson. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proc. of the International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, Sept. 1992.